Tool-supported Refactoring for JavaScript

Technical Report

Asger Feldthaus Todd Millstein Anders Møller
Max Schäfer Frank Tip

August 15, 2011

1 Syntactic Preliminaries

We assume that the program to be analyzed and refactored is represented as an abstract syntax tree (ASTs). An AST node (or just *node* for short) is characterized both by its content, i.e., a piece of program text represented by the node, and its position in the syntax tree for the entire program. For convenience, we will sometimes elide the distinction between a node and its content.

The following node types are used in this specification:

Identifier reference An identifier used as an expression [1, §11.1.2]. Its name is the *IdentifierName* with Unicode escape sequences of the form `\uxxxx` normalized to their CVs [1, §7.8.4]. The set of all identifier reference nodes in the program is denoted as `IdRef`.

Fixed-property expression A property accessor using dot notation [1, §11.2.1]. Its base expression is the expression to the left of the dot, its name the (normalized) identifier to the right. The set of fixed-property expressions is denoted as `FixedPropExp`.

Dynamic property expression A property accessor using bracket notation [1, §11.2.1]. Its base expression is the expression to the left of the opening bracket. The set of dynamic property expressions is denoted as `DynamicPropExp`.

Property initializer A property assignment in an object literal [1, §11.1.5]. Its name is the property name to the left of the colon, resp. after the `get` or `set` annotation. The set of fixed-property initializers is denoted as `PropInit`.

In expression An expression whose top-level operator is the binary `in` operator. Its base expression is the expression to the right of the `in` operator. The set of in expressions is denoted as `InExp`.

Variable declaration A variable declaration [1, §12.2]. Its name is the (normalized) identifier. The set of variable declarations is denoted as `VarDecl`.

Function definition A function expression or declaration [1, §13]. The set of function definitions is denoted as `FunDef`. For every node n, *enclosingFun*(n) denotes the innermost function definition lexically enclosing n, not counting n itself, if there is one.

Formal parameter An identifier occurring in the formal parameter list of a function definition [1, §13] or in a `catch` clause [1, §12.14]. Its name is the (normalized) identifier itself. The set of formal parameters is denoted as `FormalParm`.

Function name An identifier occurring in a function definition [1, §13]. For convenience, we assume that every function has a name; unnamed function expressions can be thought of as having a unique synthetic name. The name of a function name is the (normalized) identifier itself. The set of function names is denoted as `FunName`.

Function call The set of function calls is denoted as `FunCall`.

New expression The set of `new` expressions is denoted as `NewExp`

Invocation An invocation is either a function call or a `new` expression. The set of invocations is denoted as `Invocation`; for an invocation n, *funExp*(n) denotes the invoked expression.

For-in loop A loop of the form `for`$(h$ `in` $e)\,b$. The set of for-in loops is denoted as `ForInLoop`; for a for-in loop l, *loopExp*(l) denotes the expression being iterated over.

Catch clause The set of `catch` clauses is denoted as `CatchClause`.

With block The set of `with` blocks is denoted as `WithBlock`. For a `with` block b, *withExp*(b) denotes the expression the `with` is performed on.

Scope node A scope node is either a function definition or a `catch` clause or a `with` block.

An expression node e is in *lvalue position* if it is the left hand side of an assignment, the operand of `delete` or of one of the increment or decrement operators; it is in *rvalue position* if it is not the left hand side of a simple assignment or the operand of `delete`. Parentheses are ignored for the purposes of determining whether an expression is in lvalue or rvalue position.

Identifier references, fixed-property expressions, property initializers, variable declarations, formal parameters and function names are collectively referred to as *named accesses*. Every named access a has a name (as defined in the above description), which we write as $\nu(a)$. Named accesses, dynamic property expressions and in expressions are referred to as *accesses*.

For an access a and a name p, we say that a *may have name* p if either a is a named access and $p = \nu(a)$, or a is a dynamic property expression $e[e']$ such that e' may evaluate to a value v with $p = \texttt{ToString}(v)$, or a is an in expression $e \texttt{ in } e'$ such that e may evaluate to a value v with $p = \texttt{ToString}(v)$.

Similarly, we say that a *must have name* p if either a is a named access and $p = \nu(a)$, or a is a dynamic property expression $e[e']$ such that for every value v that e' evaluates to we have $p = \texttt{ToString}(v)$, or a is an in expression $e \texttt{ in } e'$ such that for every value v that e evaluates to we have $p = \texttt{ToString}(v)$.

We assume that there are queries *possiblyNamed* and *definitelyNamed* which, for a given property name p, determine an over- resp. under-approximation of the set of access nodes in the program that may (must) have name p. These queries can be implemented entirely syntactically, although an implementation may choose to supplement the syntactic information with additional information gained from static analysis.

Given an access a and an identifier y, we let $\rho(a, y)$ denote the access that results when replacing the name of a by y. Specifically:

1. For an identifier reference x, define $\rho(x, y) := y$.
2. For a fixed-property expression $e.x$, define $\rho(e.x, y) := e.y$.
3. For a dynamic property expression $e[e']$, define $\rho(e[e'], y) := e.y$.
4. For a property initializer $p\colon e$, define $\rho(p\colon e, y) := y\colon e$, and similar for `get` and `set` properties.
5. For an in expression $e \texttt{ in } e'$, define $\rho(e \texttt{ in } e', y) := \texttt{"}y\texttt{"} \texttt{ in } e'$.
6. For a variable declaration $\texttt{var } x$, define $\rho(\texttt{var } x, y) := \texttt{var } y$, and similar for variable declarations with initializer.
7. For a formal parameter x, define $\rho(x, y) := y$.
8. For a function name f, define $\rho(f, y) := y$.

We write $T = \langle t_1, \ldots, t_m \rangle$ for the list of top-level statements in the program under consideration, in the order in which they are evaluated by the browser or stand-alone interpreter.

We write $\mathbb{P}$ for the set of property names.

2 Central Concepts

We say that two access nodes a_1 and a_2 are *directly related* if they may resolve to the same property in some evaluation, and (indirectly) *related* if they are transitively directly related. Related access nodes always have to be renamed together in order to preserve behavior.

To capture the concept that a JavaScript function behaves "like a constructor" or "like a method", we introduce the concepts of *initialization* and *well-scopedness*.

A function definition node f initializes an object if it is invoked precisely once with that object as a receiver, and that invocation happens before any properties of the object are accessed.

A function object o_m is well-scoped if it results from evaluating a function definition node m that is a direct inner function of another function definition node f and whenever o_m is invoked its receiver object is the same as the receiver object of the invocation of f that created o_m. If every function object resulting from m is well-scoped, we say that m itself is well-scoped. Generalizing to deeper nesting, we say that m is well-scoped up to h if m is a (not necessarily direct) inner function of h, and m and every enclosing function up to h is well-scoped.

3 The Underlying Analysis

This section codifies our assumptions about the underlying pointer analysis.

We assume that the analysis defines a finite set $\mathbb{L}$ of *object labels* such that every object at runtime has a (not necessarily unique) label drawn from $\mathbb{L}$. Environment records also have labels. For technical reasons, we require that if an object label represents an object allocated by a particular `new` expression, then all objects represented by the label are allocated by that expression. Similarly, we require that the sets of object labels representing different function objects are disjoint.

Let $\mathbb{L}_f$ be the subset of object labels that may represent function objects, and $\mathbb{L}_h$ the subset of object labels that may represent host objects.

We say that $L \subseteq \mathbb{L}$ is an *over-approximation* of a set of O runtime objects if L contains all the labels of the objects in O. Conversely, L is an *under-approximation* of O if all objects labelled by any of the object labels in L are contained in O.

We expect the pointer analysis to provide the following basic queries:

objects For any expression e in the program, $\mathit{objects}(e) \subseteq \mathbb{L}$ over-approximates the set of objects to which e may evaluate, including objects arising from `ToObject` conversion [1, §9.9]. For a function declaration f, $\mathit{objects}(f)$ over-approximates the set of function objects that may result from evaluating f.

scope For any function definition or catch clause s, $\mathit{scope}(s) \subseteq \mathbb{L}$ over-approximates the set of environment records that may appear at the top of the lexical environment when the body of s is executed. We additionally define $\mathit{scope}(w) := \mathit{objects}(\mathit{withExp}(w))$ for any `with` block w.

proto For any object label ℓ, $\mathit{proto}(\ell) \subseteq \mathbb{L}$ over-approximates the possible prototype objects of the objects represented by ℓ. We write $\mathit{proto}^+(L)$ for the set of transitive prototypes of L as determined by this query, and similar for $\mathit{proto}^*(L)$.

props For any object label ℓ, $\mathit{props}(\ell) \subseteq \mathbb{L}$ over-approximates the set of objects that could be stored in properties of ℓ.

mayHaveProp, *mustHaveProp* For any object label ℓ and property name p, $\mathit{mayHaveProp}(\ell, p)$ must hold whenever any object represented by ℓ may have a property p; $\mathit{mustHaveProp}(\ell, p)$, conversely, should only hold if every object represented by ℓ must definitely have a property p.

arg, *ret* For an object label ℓ and a natural number i, $\mathit{arg}(\ell, i)$ over-approximates the set of objects that may be passed as the ith argument (or the receiver in case $i = 0$) to any function labelled by ℓ. Similarly, *ret* over-approximates the set of objects that may be returned from ℓ.

builtin Given the name of a standard built-in ECMAScript object or property (as specified in [1, §15]), this query returns the corresponding object label. We write *global* for the object label corresponding to the global object. For brevity, we use the names of built-in ECMAScript objects such as *Function.prototype.call* to stand for their corresponding object labels.

4 Derived Queries

We define the following derived queries in terms of the above:

scopes Given an expression node e and a name n, $\mathit{scopes}(e, n)$ computes an over-approximation of the set of environment records and objects which may be searched when n is looked up at position e.

Let $s_1, \ldots, s_n$ be the sequence of scope nodes enclosing e in order of nesting, with s_1 being the innermost enclosing scope node. For every $i \in \{1, \ldots, n\}$ let

$$L_i := \begin{cases} \mathit{scope}(s_i) & \text{if } s_i \in \texttt{FunDef} \cup \texttt{CatchClause} \\ \mathit{proto}^*(\mathit{objects}(\mathit{withExp}(s_i))) & \text{if } s_i \in \texttt{WithBlock} \end{cases}$$

Additionally, let $L_{n+1} := \{\mathit{global}\}$.

Let k be the smallest index such that $\mathit{mustHaveProp}(\ell, n)$ holds for every $\ell \in L_j$; if no such index exists, define $k := n + 1$.

Finally, let $\mathit{scopes}(e, x) := \bigcup_{1 \leq i \leq k} L_i$.

visited Given an access node a, $\mathit{visited}(a)$ over-approximates the set of objects that may be visited during resolution of a.

1. For an identifier reference x, $\mathit{visited}(x) := \mathit{scopes}(x, \nu(x))$.
2. For a fixed-property expression $e.x$ that is not in rvalue position, define $\mathit{visited}(e.x) := \mathit{objects}(e)$.

   If $e.x$ is in rvalue position, $\mathit{visited}(e.x)$ is the smallest set L such that

   (a) $\mathit{objects}(e) \subseteq L$

   (b) $\forall \ell \in L\colon \neg \mathit{mustHaveProp}(\ell, \nu(e.x)) \rightarrow \mathit{proto}(\ell) \subseteq L$

3. For a dynamic property expression $e[e']$ in rvalue position:

$$\mathit{visited}(e[e']) := \mathit{proto}^*(\mathit{objects}(e))$$

   If $e[e']$ is not in rvalue position:

$$\mathit{visited}(e[e']) := \mathit{objects}(e)$$

4. For a property initializer p in an object literal l, $\mathit{visited}(p) := \mathit{objects}(l)$.
5. For an in expression e `in` e', $\mathit{visited}(e \texttt{ in } e') := \mathit{proto}^*(\mathit{objects}(e'))$.
6. For a variable declaration d without initializer:

$$\mathit{visited}(d) := \begin{cases} \mathit{scope}(f) & \text{if } f := \mathit{enclosingFun}(d) \text{ is defined} \\ \{\mathit{global}\} & \text{otherwise} \end{cases}$$

   If d has an initializer and the directly enclosing scope node s of e is not a function declaration, $\mathit{visited}(d)$ additionally contains $\mathit{scope}(s)$.
7. For a formal parameter x belonging to $s \in$ `FunDef` $\cup$ `CatchClause`, $\mathit{visited}(x) := \mathit{scope}(s)$.
8. For a function name f belonging to a function definition d:

$$\mathit{visited}(f) := \begin{cases} \mathit{scope}(g) & \text{if } g := \mathit{enclosingFun}(d) \text{ is defined} \\ \{\mathit{global}\} & \text{otherwise} \end{cases}$$

We also define a filtering version of *visited* as follows:

$$\mathit{visited}(a, n) := \{\ell \in \mathit{visited}(a) \mid \mathit{mayHaveProp}(\ell, n)\}$$

This discards from $\mathit{visited}(a)$ all object labels that cannot possibly have a property n.

base Given an access node a, $\mathit{base}(a)$ approximates the set of base objects on which a may be looked up.

If a is an identifier reference node in rvalue position, a property expression node in rvalue position or an in expression, and a must have name n, define $\mathit{base}(a) := \mathit{visited}(a, n)$. Otherwise define $\mathit{base}(a) := \mathit{visited}(a)$.

lookup For an expression node e and a name n, define

$$\mathit{lookup}(e, n) := \{\ell \in \mathit{scopes}(e, n) \mid \mathit{mayHaveProp}(\ell, n)\}$$

related For an access node a, the set $\mathit{related}(a)$ is the smallest set R such that

1. $a \in R$
2. $\forall p \in \mathbb{P}\colon \forall a_1, a_2 \in \mathit{mayHaveName}(p)\colon a_1 \in R \wedge \mathit{base}(a_1) \cap \mathit{base}(a_2) \neq \emptyset \rightarrow a_2 \in R$

callees For an invocation expression c define

$$callees(c) := objects(funExp(c))$$

intrinsic For an object label ℓ and a property name p, $intrinsic(\ell, p)$ holds whenever p is an intrinsic property on some object o represented by ℓ, i.e., the runtime system, browser or standard library treat property p of object o specially.

Concretely:

1. $\forall \ell \in \mathbb{L}\colon \{\ell\} \times \{\texttt{toString}, \texttt{valueOf}, \texttt{constructor}\} \subseteq intrinsic$
2. For every $\ell \in \mathbb{L}$ such that

   $$proto(\ell) \cap \{Array.prototype, String.prototype, Function.prototype\} \neq \emptyset$$

   we let $(\ell, \texttt{length}) \in intrinsic$.
3. $\forall \ell \in \mathbb{L}_f\colon (\ell, \texttt{prototype}) \in intrinsic$.
4. $\forall p \in \mathbb{P}\colon \forall \ell \in \mathbb{L}_h\colon (\ell, p) \in intrinsic$
5. $\{global\} \times \{\texttt{document}, \texttt{window}\} \subseteq intrinsic$
6. $\{global\} \times G \subseteq intrinsic$, where
   $G = \{\texttt{NaN}, \texttt{Infinite}, \texttt{undefined}, \texttt{eval}, \texttt{parseInt}, \texttt{parseFloat}, \texttt{isNaN}, \texttt{isFinite}, \texttt{decodeURI}, \texttt{decodeURIComponent}, \texttt{encodeURI}, \texttt{encodeURIComponent}, \texttt{Object}, \texttt{Function}, \texttt{Array}, \texttt{String}, \texttt{Boolean}, \texttt{Number}, \texttt{Date}, \texttt{RegExp}, \texttt{Error}, \texttt{EvalError}, \texttt{RangeError}, \texttt{ReferenceError}, \texttt{SyntaxError}, \texttt{TypeError}, \texttt{URIError}, \texttt{Math}, \texttt{JSON}\}$

reflPropAcc For an object label ℓ, $reflPropAcc(\ell)$ holds whenever a property of an object labelled by ℓ may be accessed reflectively by a built-in function.

Concretely:

1. $arg(Object.getOwnPropertyDescriptor, 1) \subseteq reflPropAcc$
2. $arg(Object.getOwnPropertyNames, 1) \subseteq reflPropAcc$
3. $arg(Object.defineProperty, 1) \subseteq reflPropAcc$
4. $arg(Object.defineProperties, 1) \subseteq reflPropAcc$
5. $arg(Object.defineProperties, 2) \subseteq reflPropAcc$
6. $arg(Object.keys, 1) \subseteq reflPropAcc$
7. $arg(Object.prototype.hasOwnProperty, 0) \subseteq reflPropAcc$
8. $arg(Object.prototype.propertyIsEnumerable, 0) \subseteq reflPropAcc$
9. $props^*(arg(JSON.stringify, 1)) \subseteq reflPropAcc$
10. $props^*(ret(JSON.parse)) \subseteq reflPropAcc$

initializes For a function definition node f, *initializes*(f) is an under-approximation of the set of objects which f initializes.

We say that a function definition node f is an *initializer* if it initializes all of its receiver objects. One way of computing *initializes*(f) is to check whether g is an initializer; if so, we can set

$$\mathit{initializes}(f) := \bigcup_{\ell \in \mathit{objects}(f)} \mathit{arg}(\ell, 0).$$

Otherwise, we conservatively let $\mathit{initializes}(f) := \emptyset$.

The basic idea is that a function is an initializer if every potential call to f is either a `new` expression that definitely invokes f, or a "superconstructor" invocation of the form `F.call(this, ...)` that definitely invokes f and is the first statement of another function that is itself an initializer.

Let us call a function call node c a *superconstructor call* if

1. c is the expression of an expression statement, which in turn is the first statement in the body of a function definition;
2. the first argument of c is a `this` expression;
3. $\mathit{callees}(c) \subseteq \{\mathit{Function.prototype.apply}, \mathit{Function.prototype.call}\}$.

We say that a function call node c *may reflectively invoke* a function definition f if

1. $\mathit{callees}(c) \cap \{\mathit{Function.prototype.apply}, \mathit{Function.prototype.call}\} \neq \emptyset$;
2. either $\mathit{funExp}(c)$ is a property expression with base expression e such that
$$\mathit{objects}(e) \cap \mathit{objects}(f) \neq \emptyset,$$
or there is a surrounding `with` block w such that
$$\mathit{objects}(\mathit{withExp}(w)) \cap \mathit{objects}(f) \neq \emptyset.$$

We say that a function call node c *must reflectively invoke* a function definition f if

1. $\mathit{callees}(c) \subseteq \{\mathit{Function.prototype.apply}, \mathit{Function.prototype.call}\}$;
2. $\mathit{funExp}(c)$ is a property expression with base expression e such that
$$\mathit{objects}(e) \subseteq \mathit{objects}(f).$$

In order for a function definition f to be an initializer, it suffices if the following conditions hold:

1. `Function.prototype.bind` is never invoked on f:
$$\mathit{arg}(\mathit{Function.prototype.bind}, 0) \cap \mathit{objects}(f) = \emptyset$$

2. `Function.prototype.apply`, `bind` and `call` are not invoked on each other.
3. Every invocation node c such that $\mathit{callees}(c) \cap \mathit{objects}(f) \neq \emptyset$ is a `new` expression with $\mathit{callees}(n) \subseteq \mathit{objects}(f)$.
4. Every invocation node c that may reflectively invoke f must reflectively invoke f, and it must be a superconstructor call in a function f' such that f' is itself an initializer.

wellscoped For a function definition node f, $\mathit{wellscoped}(f)$ holds if f is well-scoped.

It suffices to check the following conditions:

1. f is a direct inner function of function definition g.
2. f is only assigned to properties of the receiver of g: whenever the right-hand side e_r of a simple assignment may evaluate to f (i.e., $\mathit{objects}(e_r) \cap \mathit{objects}(f) \neq \emptyset$), the sole intra-procedural reaching definition of e_r is f itself, and the left-hand side of the assignment is a property expression of the form `this`.p (for some identifier p).
3. f is only invoked on the object in whose property it is stored:
   (a) For any invocation expression c with $\mathit{callees}(c) \cap \mathit{objects}(f) \neq \emptyset$, $\mathit{funExp}(c)$ is of the form $e.p$.
   (b) $\forall \ell \in \mathit{proto}^+(\mathit{objects}(e)) \colon \neg \mathit{mayHaveProp}(\ell, p)$
4. f is not invoked reflectively.

5 Specification of Rename

Input A named access node a and an identifier y.

Overview a and its related accesses are renamed to y.

Definitions Let $B := \bigcup_{r \in \mathit{related}(a)} \mathit{base}(r)$; this set labels all objects that are affected by the renaming. Let x be the name of a.

Preconditions

1. x is not an intrinsic property on B:

$$\forall b \in B \colon (b, x) \notin \mathit{intrinsic}$$

2. Every access to be renamed definitely has name x:

$$\mathit{related}(a) \subseteq \mathit{definitelyNamed}(x)$$

3. For every `in` expression e `in` e' or dynamic property expression $e'[e]$ in $\mathit{related}(a)$, expression e' must be side effect free (since it will be replaced by the renaming).

4. The accesses in $\mathit{related}(a)$ can be renamed to y without name capture:

$$\forall r \in \mathit{related}(a) \colon \mathit{visited}(r, y) = \emptyset$$

   We also say that y is *free* for $\mathit{related}(a)$.

5. y does not cause name capture on B, that is:

   (a) Existing accesses are not captured:

$$\forall r \in \mathit{possiblyNamed}(y) \colon \mathit{visited}(r) \cap B = \emptyset$$

   (b) y is not an intrinsic property on B.

   (c) Properties of the objects in B must not be accessed reflectively, that is:

      i. $\forall f \in \texttt{ForInLoop} \colon B \cap \mathit{objects}(\mathit{loopExp}(f)) = \emptyset$
      ii. $\forall \ell \in B \colon \neg \mathit{reflPropAcc}(\ell)$.

Transformation Replace every $a \in \mathit{related}(a)$ with $\rho(a, y)$.

6 Specification of Encapsulate Property

Input A fixed-property expression node a.

Overview This refactoring identifies a function c that initializes all base objects of a and its related accesses, and turns the property accessed by a into a local variable of c.

Any accesses to the property from within c can be turned into accesses to the local variable if they happen from within well-scoped functions. Accesses from outside c are handled by defining getter and setter functions in c and rewriting accesses into calls to these functions.

The preconditions identify a suitable c, determine how to rewrite accesses, and prevent problematic name binding changes.

Definitions Let $x := \nu(a)$ be the name of a; let g and s be appropriate getter and setter names derived from x.

Let $B := \bigcup_{r \in \mathit{related}(a)} \mathit{base}(r)$; this is the set of objects whose properties x we want to encapsulate.

Preconditions

1. There is a function node c such that $B \subseteq \mathit{initializes}(c)$.

   The getter and setter functions are defined in c; since it is invoked on every affected object before any of its properties are accessed, we can be sure that the functions are in place before their first use.

2. The affected objects do not appear on each others' prototype chains, i.e.,

   $$\neg\exists b_1, b_2 \in B : b_2 \in \mathit{proto}^+(b_1)$$

3. Every access in $\mathit{related}(a)$ is either a fixed-property expression or an identifier reference.[1]

4. There is a decomposition $\mathit{related}(a) = A_i \uplus A_g \uplus A_s$ such that

   (a) Every $a \in A_i$ is of the form `this.x`, is not an operand of `delete`, and its enclosing function is well-scoped up to c.

   These are the accesses that will be replaced by identifier references `x`. Well-scopedness ensures that this reference refers to the right `x`.

   (b) None of the $a \in A_g$ is in an lvalue position.

   These accesses can be turned into invocations of the getter function.

   (c) Every $a \in A_s$ is in an lvalue position, but not an operand of `delete`. If it is a property expression that does not form the left-hand side of a simple assignment expression, its base expression must be side effect-free. If it is the operand of a post-decrement/-increment operator, the whole expression (including the operator) must either be the left operand of a comma operator, or the operand of a `void` operator, or the expression of an expression statement.

   These accesses will be turned into setter invocations if they form the left-hand side of a simple assignment expression, or a combination of getter and setter invocations otherwise. In the latter case, the base expression has to be duplicated, hence the condition about side effects. The effect of post-decrement and post-increment is hard to mimick using getters and setters, except if their resulting value is not observable, in which case they act like pre-decrement/-increment.

   Let $A_r := \{a' \in \mathit{related}(a) \setminus A_i \mid a' \text{ is in an rvalue position}\}$ be the set of all rvalue-expressions that are rewritten to use the getter function.

5. Properties of B must not be accessed reflectively.

6. Naming checks:

   (a) A_i can be refactored without name capture:

   $$\forall a \in A_i : \mathit{base}(a, x) = \emptyset$$

[1] The latter can only happen if a `with` statement is involved.

(b) The declaration of the new local variable `x` in c does not capture existing identifier references.

$$\forall a \in \mathit{possiblyNamed}(x) \colon \mathit{visited}(a) \cap \mathit{scope}(c) = \emptyset$$

(c) x is not an intrinsic property on B.

7. If $A_r \neq \emptyset$, g must be free for A_g and must not cause name capture on $\mathit{initializes}(c)$.

8. If $A_s \neq \emptyset$, s must be free for A_s and must not cause name capture on $\mathit{initializes}(c)$.

Transformation

1. Insert a declaration `var x;` into c.

2. If $A_r \neq \emptyset$, insert definition

```
this.g = function() {
    return x;
};
```

into c.

3. If $A_s \neq \emptyset$, insert definition

```
this.s = function(x′) {
    return x = x′;
}
```

into c, where x' is a suitable identifier derived, but different, from x.

4. For every access $a \in A_i$:

   (a) If a is the function expression of a function call with arguments $e_1, \ldots, e_n$, replace the whole function call with x.`call`(`this`, $e_1, \ldots, e_n$).[2]

   (b) Otherwise, replace a with an identifier reference x.

5. For every access $a \in A_g$: if a is a name expression, replace it with $g()$; otherwise it must be a fixed-property expression $b.x$, which is replaced with $b.g()$.

6. For every access $a \in A_s$:

   (a) If a is the left-hand side of a simple assignment expression with right-hand side e, replace it with $s(e)$ or $b.s(e)$, depending on whether it is a name expression or a fixed-property expression $b.x$.

[2] We assume that `Function.prototype.call` has not been redefined.

(b) If a is the left-hand side of a composite assignment expression with operator $\circ$, replace it with $s(g() \circ e)$ or $b.s(b.g() \circ e)$, depending on whether it is a name expression or a fixed-property expression $b.x$.

(c) Otherwise, a is either the operand of an increment operator, in which case replace it with $s(g() + 1)$ or $b.s(b.g() + 1)$, or the operand of a decrement operator, which is handled analogously.

7 Specification of Extract Module

Input Contiguous top-level statements $t_k, \ldots, t_{k+l}$ occurring in the same `script` element or source file, a set P of identifiers to extract and an identifier M to be used as module name.

Overview Schematically, the transformation performed by the refactoring is as follows (where we write $p_1, \ldots, p_n$ for the identifiers in P):

t_k;
⋮
t_{k+l};

$\Rightarrow$

```
var M = (function() {
  var p1, ..., pn;
  tk; ... tk+l;
  return {
    p1: p1, ..., pn: pn
  };
})();
```

The extracted global variables $p_1, \ldots, p_n$ are turned into properties of the newly declared module variable M; hence they can be accessed as $M.p_1, \ldots, M.p_n$ after module initialization has finished, i.e., the definition of M has been evaluated. There should not be any accesses to these variables *before* module initialization, and any accesses *during* module initialization have to instead refer to the corresponding local variables $p_1, \ldots, p_n$.

If some p_i is not assigned to after module initialization, accesses lexically nested inside the module can access the local variable instead and do not need qualification with M.

The preconditions determine a set Q of accesses that have to be qualified, and a set U of accesses that do not; in addition, they ensure that there are no accesses before module initialization and prevent name binding issues.

Definitions Let S be the set of all access nodes appearing in the given statements, with its subset I containing only those nodes not nested inside functions. Accesses in I are thus guaranteed to only be evaluated during module initialization.

Let I^* be an overapproximation of the set of all access nodes that may be evaluated before or during module initialization. This can be computed by building a transitive call-graph of the statements $t_1, \ldots, t_{k+l}$, using query *callee* to determine possible callees of invocation nodes. Finally, let C contain all

access nodes not in I^*; accesses in C are thus guaranteed not to be evaluated before module initialization is complete.

For $p \in P$, we define $A_p := \{a \in \mathit{possiblyNamed}(p) \mid \mathit{global} \in \mathit{base}(a, \rho(a, p))\}$ to be the set of accesses that may refer to the global property p, and $A_P := \bigcup_{p \in P} A_p$. We define $\mathit{mutable}(p)$ to hold if A_p contains a write access that does not belong to I, i.e., if p may be written after module initialization is complete.

Preconditions

1. Any access that *may* refer to some property in P *must* refer to that property; i.e., for every $p \in P$ and $a \in A_p$:

$$a \in \mathit{definitelyNamed}(p) \land \mathit{visited}(a, p) = \{\mathit{global}\}$$

2. There is a partitioning $A_P = Q \uplus U$ into accesses that will be qualified with M and accesses that will be turned into identifier references.

   Q must fulfill the following conditions:

   (a) $Q \subseteq C$, i.e., accesses in Q can only be evaluated after module initialization is finished.

   (b) M is free for Q.

   (c) For any identifier reference in Q that is the invoked expression of a function call c, no possible callee of c may reference `this`. That is, there may not be any `this` expression t such that

$$\mathit{objects}(\mathit{enclosingFun}(t)) \cap \mathit{callees}(c) \neq \emptyset.$$

   (d) For any fixed-property expression $e.f$ in Q that is the invoked expression of a function call, e must be side effect-free (since it will be duplicated by the transformation).

   Every $u \in U$ referring to $p \in P$ must fulfill the following conditions:

   (a) $u \in I \lor u \in S \cap \neg \mathit{mutable}(p)$

   (b) $\mathit{base}(u, p) \subseteq \{\mathit{global}\}$.

   (c) If u is not an identifier reference, it must be a fixed-property expression of the form $e.p$ such that e is side-effect free.

3. M does not cause name capture on *global*.

4. No $p \in P$ is an intrinsic on *global*.

Transformation

1. Replace $t_k, \ldots, t_{k+l}$ with the definition of module M as shown above.
2. For every access $q \in Q$:
   (a) If q is an identifier reference, replace it with $M.q$.
   (b) If q is a fixed-property expression $e.p$ that forms the function expression of a function call node with arguments $e_1, \ldots, e_n$, replace it with $e.M.p.\texttt{call}(e, e_1, \ldots, e_n)$. If it does not form such a function expression, replace it with $e.M.p$.
   (c) Otherwise, q must be a variable declaration declaring a variable p. If q has an initializer expression e, turn it into an assignment $M.p = e$. Otherwise delete q.
3. For every access $u \in U$: if u is a fixed-property expression, replace it with an identifier reference formed from its name.

8 Specification of Inline Function

Inlining a function invocation in JavaScript can be done in three steps:

1. Inline the called function into a *one-shot* closure, i.e., a function expression that is immediately applied.
2. Turn arguments into local variables.
3. Convert the closure into a block and inline the local variables.

We concentrate on the first step only, even though it is not very useful as a standalone refactoring.

Input A function invocation node f of the form $e_0(e_1, \ldots, e_n)$.

Overview Once it has been determined that f can only call one unique function, the main challenge is verifying which closure variables may be accessed by the inlined function. Even if a referenced variable is in scope at f, it is possible that this variable could be read from a different lexical environment representing the same scope.

Preconditions

1. There is a function definition node c with $\mathit{objects}(e_0) \subseteq \mathit{objects}(c)$.
   This guarantees that f invokes a unique user-defined (as opposed to built-in) function.
2. For every identifier reference v in c and its inner functions, at least one of the following conditions must hold:

(a) v refers to a global variable and is not captured at f

$$base(v) = \{global\} \wedge base(f, v) = \{global\}$$

(b) v refers to a local variable of c or one of its inner functions

(c) c is a function declaration with name g, e_0 is an identifier reference `g` that refers to c (hence they share a lexical environment), and $base(v) = base(f, v)$

3. If e_0 is not a property expression, the body of c may not contain a `this` expression.

4. If e_0 is not a property expression, it must not have side effects, as they would be removed by the inlining.

Transformation Let $p_1, \ldots, p_n$ be the parameters of c and b its body. If e_0 is a property expression with receiver expression e_r, replace f by[3]

`(function(`p_1`, ..., `p_n`) { `b` }).call(`e_r`, `e_1`, ..., `e_n`)`

Otherwise, replace f by

`(function(`p_1`, ..., `p_n`) { `b` })(`e_1`, ..., `e_n`)`

References

[1] ECMA. ECMAScript Language Specification, 5th edition, 2009. ECMA-262.

[3] This transformation may be unsound if user code has redefined `Function.prototype.call`; we choose to ignore this possibility.